Lecture 20: Devices and I/O

Computer Architecture and Systems Programming
(252-0061-00)

Timothy Roscoe
Herbstsemester 2012
Last time: P6 (ia32) Address Translation

CPU

VPN

VPO

TLBT

TLBI

TLB (16 sets, 4 entries/set)

VPN1

VPN2

PDE

PTE

Page tables

L1 (128 sets, 4 lines/set)

PDBR

CT

CI

CO

physical address (PA)

virtual address (VA)

TLB (16 sets, 4 entries/set)

32

result

L1 hit

L1 miss

Page tables

L2 and DRAM

CPU

VPN

VPO

TLBT

TLBI

TLB miss

VPN1

VPN2

PDE

PTE

Page tables

L2 and DRAM

CPU

VPN

VPO

TLBT

TLBI

TLB (16 sets, 4 entries/set)

VPN1

VPN2

PDE

PTE

Page tables

L2 and DRAM

CPU

VPN

VPO

TLBT

TLBI

TLB (16 sets, 4 entries/set)

VPN1

VPN2

PDE

PTE

Page tables

L2 and DRAM

CPU

VPN

VPO

TLBT

TLBI

TLB (16 sets, 4 entries/set)

VPN1

VPN2

PDE

PTE

Page tables

L2 and DRAM

CPU

VPN

VPO

TLBT

TLBI

TLB (16 sets, 4 entries/set)

VPN1

VPN2

PDE

PTE

Page tables

L2 and DRAM

CPU

VPN

VPO

TLBT

TLBI

TLB (16 sets, 4 entries/set)

VPN1

VPN2

PDE

PTE

Page tables

L2 and DRAM

CPU

VPN

VPO

TLBT

TLBI

TLB (16 sets, 4 entries/set)

VPN1

VPN2

PDE

PTE

Page tables

L2 and DRAM

CPU

VPN

VPO

TLBT

TLBI

TLB (16 sets, 4 entries/set)

VPN1

VPN2

PDE

PTE

Page tables

L2 and DRAM

CPU

VPN

VPO

TLBT

TLBI

TLB (16 sets, 4 entries/set)

VPN1

VPN2

PDE

PTE

Page tables

L2 and DRAM

CPU

VPN

VPO

TLBT

TLBI

TLB (16 sets, 4 entries/set)

VPN1

VPN2

PDE

PTE

Page tables

L2 and DRAM

CPU

VPN

VPO

TLBT

TLBI

TLB (16 sets, 4 entries/set)

VPN1

VPN2

PDE

PTE

Page tables

L2 and DRAM

CPU

VPN

VPO

TLBT

TLBI

TLB (16 sets, 4 entries/set)

VPN1

VPN2

PDE

PTE

Page tables

L2 and DRAM

CPU

VPN

VPO

TLBT

TLBI

TLB (16 sets, 4 entries/set)

VPN1

VPN2

PDE

PTE

Page tables

L2 and DRAM

CPU

VPN

VPO

TLBT

TLBI

TLB (16 sets, 4 entries/set)

VPN1

VPN2

PDE

PTE

Page tables

L2 and DRAM

CPU

VPN

VPO

TLBT

TLBI

TLB (16 sets, 4 entries/set)

VPN1

VPN2

PDE

PTE

Page tables
Last time: VAX translation

- Virtual address requested (in seg. Px: user space)
- Virtual address of PTE (in seg. S0: system space)
- Physical address of PTE mapping the PTE we want
- Physical address of PTE we want
- The PTE: physical address of value we want
- Memory value (at last!)
Las time: Large Pages

- 4MB on 32-bit, 2MB on 64-bit
- Simplify address translation
- Useful for programs with very large, contiguous working sets
  - Reduces compulsory TLB misses
- How to use (Linux)
  - `hugetlbfs` support (since at least 2.6.16)
  - Use `libhugetlbfs`
    - `{m,c,re}alloc` replacements
One iteration of a matrix-matrix multiply:

- Each row is on different page
- More rows than TLB entries: TLB thrashing
- Solution: buffering = copy block to contiguous memory
  - $O(B^2)$ cost for $O(B^3)$ operations

Assume > 4 KB = 512 doubles
Today: I/O Devices

• What is a device?
• Registers
  – Example: NS16550 UART
• Interrupts
• Direct Memory Access (DMA)
• PCI (Peripheral Component Interconnect)
• Summary
What is a device?

Specifically, to an OS programmer:

- Piece of hardware visible from software
- Occupies some location on a bus
- Set of registers
  - Memory mapped or I/O space
- Source of interrupts
- May initiate Direct Memory Access transfers
Today: I/O Devices

• What is a device?
• Registers
  – Example: NS16550 UART
• Interrupts
• Direct Memory Access (DMA)
• PCI (Peripheral Component Interconnect)
• Summary
Registers

• CPU can load from device registers:
  – Obtain status info
  – Read input data

• CPU can store to device registers:
  – Set device state and configuration
  – Write output data
  – Reset states
Example: National Semiconductor ns16550 UART

• *Universal Asynchronous Receiver/Transmitter*
  – RS-232 serial line
• Basic IBM PC serial port
  – Chip now integrated into Southbridge on all PCs
  – Still used for PC servers
  – Rarely seen on laptops any more 😞
• First device driver typically written for a new OS...
Registers

• Details of registers given in chip “datasheets” or “data books”

• Information is rarely trusted by OS programmers 😊

From the data sheet for the ns16550 UART

8.4 LINE STATUS REGISTER

This register provides status information to the CPU concerning the data transfer. Table II shows the contents of the Line Status Register. Details on each bit follow.

Bit 0: This bit is the receiver Data Ready (DR) indicator. Bit 0 is set to a logic 1 whenever a complete incoming character has been received and transferred into the Receiver Buffer Register or the FIFO. Bit 0 is reset to a logic 0 by reading all of the data in the Receiver Buffer Register or the FIFO.

Bit 1: This bit is the Overrun Error (OE) indicator. Bit 1 indicates that data in the Receiver Buffer Register was not read by the CPU before the next character was transferred into the Receiver Buffer Register, thereby destroying the previous character. The OE indicator is set to a logic 1 upon detection of an overrun condition and reset whenever the CPU reads the contents of the Line Status Register. If the FIFO mode data continues to fill the FIFO beyond the trigger level, an overrun error will occur only after the FIFO is full and the next character has been completely received in the shift register. OE is indicated to the CPU as soon as it happens. The character in the shift register is overwritten, but it is not transferred to the FIFO.

Bit 2: This bit is the Parity Error (PE) indicator. Bit 2 indicates that the received data character does not have the correct even or odd parity, as selected by the even parity
Addressing registers

1. Memory mapped:
   - Registers appear as memory locations
   - Access using loads/stores (movb/movw/movl/movq)

2. “I/O instructions”:
   - Different (16 bit) address space for older I/O devices
   - Specific (these days) to x86 architecture
   - Special instructions: inb, outb, etc.

3. Indirection:
   - Write an “index” register with an offset, then a “data” register with the actual register value
   - Used to save scarce address space (usually I/O space)
## ns16550 Registers (each 8 bits)

<table>
<thead>
<tr>
<th>Addr.</th>
<th>Name</th>
<th>Description</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>RBR</td>
<td>Receive Buffer Register (read only)</td>
<td>DLAB=0</td>
</tr>
<tr>
<td>0</td>
<td>THR</td>
<td>Transmit Holding Register (write only)</td>
<td>DLAB=0</td>
</tr>
<tr>
<td>1</td>
<td>IER</td>
<td>Interrupt Enable Register</td>
<td>DLAB=0</td>
</tr>
<tr>
<td>2</td>
<td>IIR</td>
<td>Interrupt Identification Register (read only)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>FCR</td>
<td>FIFO Control Register (write only)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>LCR</td>
<td>Line Control Register</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>MCR</td>
<td>MODEM Control Register</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>LSR</td>
<td>Line Status Register</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>MSR</td>
<td>MODEM Status Register</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>SCR</td>
<td>Scratch Register</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>DLL</td>
<td>Divisor Latch (LSB)</td>
<td>DLAB=1</td>
</tr>
<tr>
<td>1</td>
<td>DLM</td>
<td>Divisor Latch (MSB)</td>
<td>DLAB=1</td>
</tr>
</tbody>
</table>

**DLAB = bit 7 of the LCR register**
ns16550 LSR: Line Status Register

0 7

DR OE PE FE BI THRE TEMT ERRF

- Error in Receiver FIFO
- Transmitter Empty
- Transmit Holding Register Empty
- Break Interrupt
- Framing Error
- Parity Error
- Overrun Error
- Data Ready
#define UART_BASE 0x3f8
#define UART_THR (UART_BASE + 0)
#define UART_RBR (UART_BASE + 0)
#define UART_LSR (UART_BASE + 5)

void serial_putchar(char c)
{
    // Wait until FIFO can hold more chars
    while((inb(UART_LSR) & 0x20) == 0);
    // Write character to FIFO
    outb(UART_THR, c);
}

char serial_getchar()
{
    // Wait until there is a char to read
    while((inb(UART_LSR) & 0x01) == 0);
    // Read from the receive FIFO
    return inb(UART_RBR);
}
Very simple UART driver

• Actually, far too simple!
  – But this is how the first version always looks...
• No initialization code, no error handling.
• Uses *Programmed I/O* (PIO)
  – CPU explicitly reads and writes all values to and from registers
  – All data must pass through CPU registers
• Uses *polling*
  – CPU polls device register waiting before send/receive
    • Tight loop!
  – Can’t do anything else in the meantime
    • Although could be extended with threads and care...
  – Without CPU polling, no I/O can occur
Registers are not memory

Device registers don’t behave like RAM!

• Register contents change without writes from CPU
  – Status words
  – Incoming data

• Writes to registers are used to trigger actions
  – Sending data
  – Resetting state machines
Dealing with caches

• Reads can’t come from the cache
  – Register value changes ⇒ cache becomes inconsistent
• Write-back caches (and write buffers) cause problems
  – You don’t know when the line will be written
• Reads and writes cannot be combined into cache lines
  – Registers might require single word or byte writes only
  – Line-size writes stomp on other registers
  – Even spurious reads trigger device state changes
⇒ Device register access must bypass the cache
  – Handled in the MMU: PTEs have “no cache” flag
  – I/O space access isn’t cached anyway
Other challenges

1. How to avoid polling all the time?
   - How does the CPU know when the device is ready, or finished?

2. How to avoid the CPU copying all the data?
   - Can you transfer data without going through the CPU and caches?
   - Can the CPU get on with something else?

3. Where do these register locations come from?
   - How can the OS find devices in the physical address space?
   - How are the physical addresses allocated?
Today: I/O Devices

• What is a device?
• Registers
  – Example: NS16550 UART
• Interrupts
• Direct Memory Access (DMA)
• PCI (Peripheral Component Interconnect)
• Summary
Avoiding polling

• CPU can’t poll every device to see if it’s ready
  – Waste of time
  – Takes too long to react

• Solution: device can *interrupt* the processor
  – Acts as an exception: saves state and jumps to kernel address
  – Info about source encoded in the *vector*
Interrupts

- **CPU Interrupt-request line** triggered by I/O device
  - Might be edge- or level-triggered

- **Interrupt handler** receives interrupts

- **Maskable** to ignore or delay some interrupts

- Interrupt vector to dispatch interrupt to correct handler
  - Based on priority
  - Some **nonmaskable**

- Interrupt mechanism also used for exceptions
## Recall x86 Exception Vectors

<table>
<thead>
<tr>
<th>Vector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Divide error</td>
</tr>
<tr>
<td>1</td>
<td>Debug exception</td>
</tr>
<tr>
<td>2</td>
<td><strong>Non-maskable interrupt (NMI)</strong></td>
</tr>
<tr>
<td>3</td>
<td>Breakpoint</td>
</tr>
<tr>
<td>4</td>
<td>Overflow</td>
</tr>
<tr>
<td>5</td>
<td>Bounds check</td>
</tr>
<tr>
<td>6</td>
<td>Invalid opcode</td>
</tr>
<tr>
<td>7</td>
<td>Coprocessor not available</td>
</tr>
<tr>
<td>8</td>
<td>Double fault</td>
</tr>
<tr>
<td>9</td>
<td>Coprocessor segment overrun (386 or earlier)</td>
</tr>
<tr>
<td>A</td>
<td>Invalid task state segment</td>
</tr>
<tr>
<td>B</td>
<td>Segment not present</td>
</tr>
<tr>
<td>C</td>
<td>Stack fault</td>
</tr>
<tr>
<td>D</td>
<td>General protection fault</td>
</tr>
<tr>
<td>E</td>
<td>Page Fault</td>
</tr>
<tr>
<td>F</td>
<td>Reserved</td>
</tr>
<tr>
<td>10</td>
<td>Math fault</td>
</tr>
<tr>
<td>11</td>
<td>Alignment check</td>
</tr>
<tr>
<td>12</td>
<td>Machine check</td>
</tr>
<tr>
<td>13</td>
<td>SIMD floating point exception</td>
</tr>
<tr>
<td>14-1F</td>
<td>Reserved to Intel</td>
</tr>
<tr>
<td>20-FF</td>
<td><strong>Available for external interrupts</strong></td>
</tr>
</tbody>
</table>
Programmable Interrupt Controllers

x86 CPU

NMI
INTR
INTA
D0 ...
D7

PIC

IRQ0
IRQ1
IRQ<n-1>

Device 1
Device 2
...
Device n
Today: I/O Devices

- What is a device?
- Registers
  - Example: NS16550 UART
- Interrupts
- Direct Memory Access (DMA)
- PCI (Peripheral Component Interconnect)
- Summary
Direct Memory Access

• Avoid *programmed I/O* for lots of data
  – E.g. fast network or disk interfaces
• Requires *DMA controller*
  – Generally built-in these days
• Bypasses CPU to transfer data directly between I/O device and memory
  – Doesn’t take up CPU time
  – Can save memory bandwidth
  – Only one interrupt per transfer
Old-style DMA transfer

1. Device driver requests data transfer of S bytes from disk controller; specifies buffer address A and size S to DMA controller

2. Disk controller starts DMA transfer of S bytes

3. Disk controller initiates bus request for each data byte from disk

4. DMA controller transfers byte to address A, increments A, decrements S.

5. When S == 0, DMA controller interrupts CPU to indicate transfer complete
Key DMA advantage

• **Decoupling** of data transfer from processing
  – CPU does not need to copy data to/from device
  – Doesn’t pollute CPU cache
  – Can be processed when the CPU (or OS) decides
  – Higher-performance: CPU and device work in parallel
    • Possible bus contention during transfers

• Possible disadvantages:
  – Higher setup overhead for very small transfers
  – Generally not a problem (even the UARTs do DMA!)
DMA and Caches

- DMA means memory becomes **inconsistent** with CPU caches

- Options:
  1. CPU can map DMA buffers non-cacheable
     ⇒ large hit – probably wants to process data anyway
  2. Cache can “snoop” DMAC bus transactions
     (but doesn’t scale beyond small SMP systems)
  3. OS can explicitly flush/invalidate cache regions
     ⇒ cache management important part of device drivers!
DMA and Virtual Memory

• DMA addresses are **physical**
  – Appear on external bus
• User and OS code deal with **virtual** addresses (mostly)
• OS (and device drivers) must manually translate virtual ↔ physical addresses when programming DMA controllers
  – This can require more than just a hardware page table!
  – DMA of a single virtual address region might **not be contiguous** in physical address space
  – **Scatter-gather** DMA controllers: DMA to/from a list of regions
• Very new systems: provide an IOMMU
  – Works like an MMU, but for DMA writes from devices
  – Must still be programmed by OS to match MMU state
  – Has all kinds of other interesting uses – beyond scope of this course!
Today: I/O Devices

• What is a device?
• Registers
  – Example: NS16550 UART
• Interrupts
• Direct Memory Access (DMA)
• PCI (Peripheral Component Interconnect)
• Summary
Where are all the registers?

• How does the OS know which devices exist?

• Where are the device registers in the physical address space?
  – And which interrupt vectors correspond to them?

• Solution: built into the *I/O bus design*
  – Example: PCI
PCI is...

Peripheral Component Interconnect

- An electrical standard for connecting devices
  - As is PCMCIA, PCI-X, PCI-Express, etc.
- A standard for physical connectors
- A set of “bus protocols” for communication between devices
- A software-visible interface to I/O hardware

PCle has succeeded PCI, but extends the same software-visible interface
PCI tries to solve many problems:

• Device discovery
  – Finding out which devices are in the system
• Address allocation
  – Which addresses should each device’s registers appear at?
• Interrupt routing
  – Which interrupt signals from the device should map to which exception vectors?
• Intelligent DMA
  – “Bus mastering” devices no longer need a DMA controller
Physical connections: PCI is a tree

- PCI bridges form a hierarchy
- Also:
  - PCI-ISA
  - PCI-USB
  - PCI-SCSI
  - Etc.
PCI address space is **flat**

- Each PCI device asks for a set of address ranges
  - Physical address space (32-bit or 64-bit)
  - I/O address space (usually 16-bit)
- Bridges up the tree remap addresses to the device
- Result:
  - Each device appears as a set of contiguous address ranges
    - In memory space
    - In I/O space (on x86 only)
PCI devices are self-describing

- Each device has a configuration header
  - Accessed through parent bridge, initially
- Some of the fields:

<table>
<thead>
<tr>
<th>Bits</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>Manufacturer ID (identifies Intel, 3Com, NVidia, etc.)</td>
</tr>
<tr>
<td>16</td>
<td>Model ID (specific to manufacturer)</td>
</tr>
<tr>
<td>24</td>
<td>Class code (what kind of device is this?)</td>
</tr>
<tr>
<td>8</td>
<td>Version identifier</td>
</tr>
</tbody>
</table>

- Plus:
  - Allocated/required address ranges (BAR values)
  - Interrupts
  - Electrical information
  - Etc.
Finding all the devices

• Find the PCI “root complex” bridge
  – PCI bridge at the top of the tree
  – Large PCs can have more than one

• Read configuration to find all attached devices
  – Add to the list of devices and functions
  – Record requirements for address space
  – If a bridge, recurse!

• Result is:
  – List of all devices in system, with address space requirements
Allocating addresses

- Find address ranges for each device and bridge
  Requirements include:
  - Each device has the size of address ranges it needs
  - All devices “below” a bridge have ranges that fit into the bridge’s range
  - Each bridge has a range which includes all it’s “children”.
  - Each range is aligned to some power-of-2 boundary

- Then program:
  - Each PCI bridge with translation information
  - Each device with “base-address/range” (BAR) registers
PCI Interrupts

• Four interrupt lines
  – INTA, INTB, INTC, INTD...
  – Bridges allow arbitrary wiring of device lines to bridge lines
  – Translated by root bridge into system interrupt

• PCI Express introduces MSIs
  – Message-signalled interrupts
  – Interrupt encoded as PCI write to specified address range
  – Translated by root bridge into system interrupt
  – Interrupts can be individually steered to particular cores/APICs
DMA over PCI

• PCI allows **Bus Mastering**
  – Device can issue read/write transactions to anywhere in memory
  – Even (in some cases) other PCI devices

• External DMA controllers no longer relevant
  – Controller effectively integrated with device itself
  – Principle applies: device DMAs data to/from memory
  – Much more flexibility / intelligence possible
Today: I/O Devices

• What is a device?
• Registers
  – Example: NS16550 UART
• Interrupts
• Direct Memory Access (DMA)
• PCI (Peripheral Component Interconnect)
• Summary
Intelligent devices

• Bus mastering, plus plenty of address space now

• Devices can now autonomously access any:
  – Main memory
  – Other devices

• Allows complex protocols for CPU↔Device interaction
  – Try to keep both CPU and device busy during high load
  – Extensive in-RAM buffering
  – “Descriptor rings” exchange requests and responses
Example: Intel e1000 PCI-Express Ethernet card

- Multiple send queues
- Multiple receive queues
- Hashing of packet headers to queues
- Directing interrupts to different cores
- Packet checksumming in hardware
- etc.
Summary

• Devices and the CPU communicate via:
  – Memory mapped and I/O space registers
  – Interrupts and interrupt vectors
  – Direct Memory Access (DMA)

• I/O Interconnects (such as PCI):
  – Allow devices to share/allocate physical addresses
  – Allocate interrupts
  – Permit bus mastering direct memory access